

CS 152: Programming Languages

`http://www.fas.harvard.edu/~cs152`

Time and Place: MWF 11:00–12:00, MD G135

Email: `cs152@fas.harvard.edu`

Course staff: Norman Ramsey, MD 231

Christian Carrillo

Jon Eddy

(MD \equiv Maxwell Dworkin)

What are programming languages for?

Express computations

- precisely,
- at a high level,
- in a way we can reason about them.

Why study programming languages?

Learn new ways of thinking about programming
language shapes thought —Whorf

Writing programs is fundamental

Learn how language can help or hinder

Become a sophisticated, skeptical consumer

CS 152 Agenda

Intellectual tools to understand & evaluate languages

- Language features
- Questions with answers

Learn the ***notations of the trade***

- Precise way to model languages
- Foundation for further study

Learn by doing

- Write lots of (mostly short) programs
- Many difficult programs (thought required)
(High difficulty per line of code)

Study of Language \equiv Study of Features

Language features influences code (**Whorf**)

Choose abstractions (languages) to fit needs

Build your vocabulary (add to your toolbox)

Higher-order functions

Polymorphism (reuse)

Pattern matching for symbolic computing

Data for symbolic computing: lists, tables, sets

Abstract datatypes, encapsulation

Objects and subtyping

Modules, parameterization

Searching and backtracking

How to use Features

From the definition of Scheme:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

(Larry Wall and Bjarne Stroustrup might not agree.)

Why study weird features?

Some languages more powerful than others

Mistake to use any but the most powerful

Except for: compatibility, libraries

Problem: habit blinds us to power

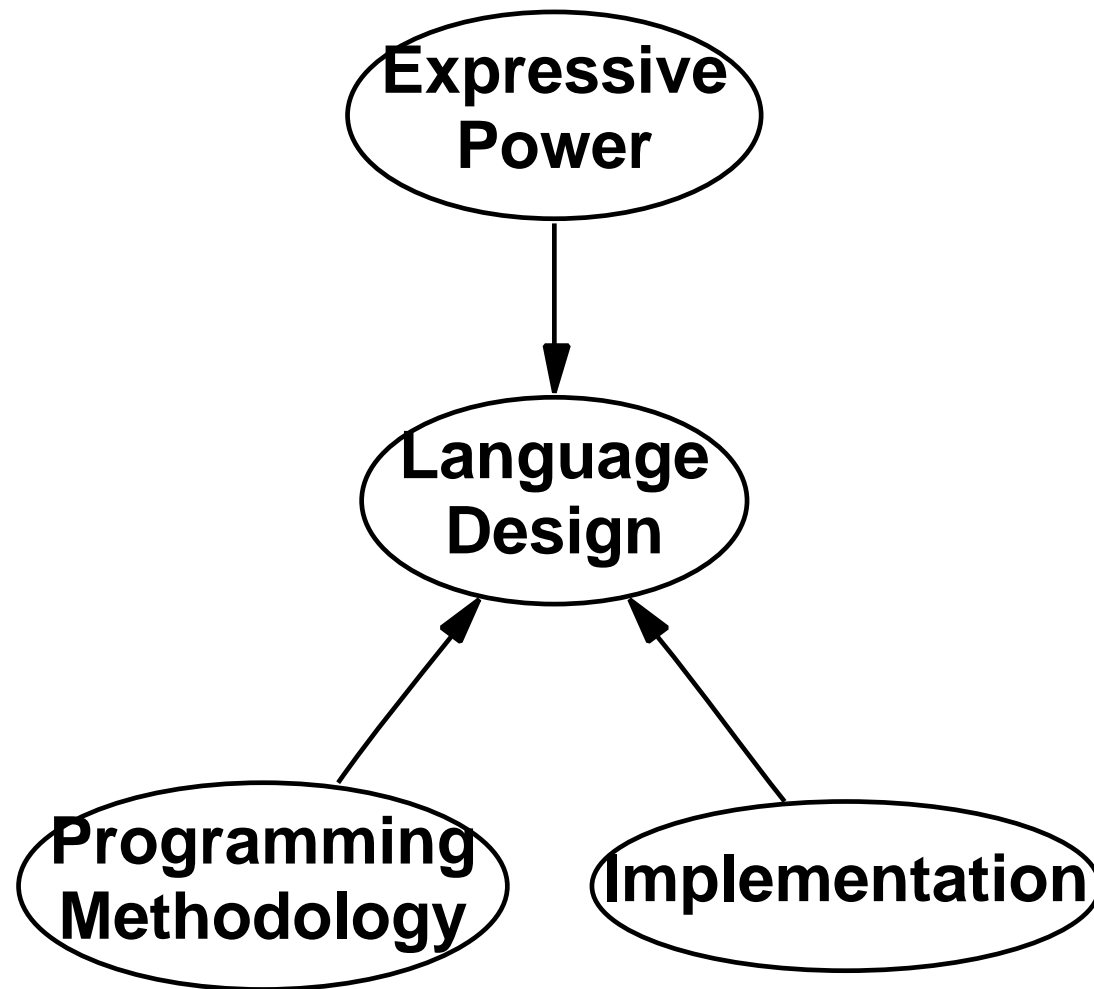
Happy user of *Blub*: beats Cobol, machine code

Use **Haskell, Lisp, or Icon**? No! Equivalent to *Blub*, plus weird stuff nobody uses

Blub looks good enough because **I think in *Blub***

CS 152: tour of power in languages

Course orientation: Language Design



The search for expressive power

Functional programming

Type systems

Influence of Methodology

Programming methodologies, software engineering

Abstract data types

Modules (including “generics”)

Objects and inheritance (reuse)

Separate compilation/smart recompilation

Influence of Implementation

Techniques

Parser generators

Attribute-grammar systems

Memory allocation

Garbage collection

Runtime typing/tagging

Efficiency concerns

fast execution

fast compilation

fast program construction

(Primary topic of CS 153)

Describing it all precisely

Formal semantics:

Operational semantics (tool of the trade)

Denotational semantics (for mathematicians)

Axiomatic semantics

Predicate transformers

Some design dimensions

Typing

strong vs. weak (ill-defined terms)

static

dynamic

polymorphic

First-class values

structures?

procedures? (funarg problem)

are built-in types different?

More design dimensions

Safety

no *unexplained* core dumps (and more ...)

Control flow

stack-based

heap-based, closures & continuations

logic programming (Prolog, unification)

backtracking (Icon)

Non-dimensions:

“Simplicity,” “Orthogonality,” “Readability”

...

Administrivia — Grading

Weight of grades:

homeworks/projects	55%
term paper	15%
midterm exam	10%
final exam	20%

Weights may be adjusted at instructor's discretion.

Administrivia — Homework

About 1 per week

Readability counts!

Most submitted electronically (at **midnight**)

Up to 10 late days for homework (≤ 48 hours per)

Course staff goals:

- Graded 7 days after 48-hour deadline
- Read and evaluated by a human being

Student can request regrade within 7 days

Administrivia — Working together

Collaborate! (Up to a point)

- what professionals do
- **vital to your success**
- discuss problems, techniques, ideas
- all discussions *must* be **acknowledged**

Must not collaborate on code

- must ***not even see*** another student's code

Seek answers in the library

- must be acknowledged
- don't overuse

Administrivia — Policies, procedures

Policies and procedures

- **handed out in class**
- **on the web**

Know what is expected

Administrivia — Computing, communication

Course run on FAS Unix servers (“ice”)

Class discussion and questions

- questions to `cs152@fas`
- answers, discussion on newsgroup
`harvard.course.cs152`

Do *not* send email directly to course staff

Prerequisites

Very good programming skills

“You must also like math”

C (or C++)

Unix

**Basic mathematics (set theory, logic, induction)
(satisfied by 121, ...)**

Course of Study

Work hard, learn a lot

Focus on

- **semantics, not syntax**
- **the unusual, not the common (weird but powerful)**
- ***answerable* questions**

Methods of study

Case studies of interpreters

- Learn foundations of languages by **studying and modifying implementations**
- Study **abstracted “essentials”** of languages
- (Mostly) uniform implementation framework

Supplement by

- **Descriptive tools** of the professionals:
 - λ -calculus
 - type systems
 - operational and denotational semantics
- **Writing**

Readings

Ramsey and Kamin

distilled essence of languages

uniform syntax, implementation framework

Standard ML

major functional language

remarkable ideas

efficient implementation

Ullman's text: intro for C programmers

Allison

Theory for the practitioner

Denotational semantics as a designer's tool

Cardelli — master of type systems

Syllabus

Unit/Language *Concepts*

Imp. Core

**environments, bindings, ASTs,
operational semantics**

Scheme

**S-expressions, recursion and lists,
programs as data, first-class &
higher-order functions**

Memory mgmt

garbage collection

ML

polymorphic typing, type inference

λ -calculus

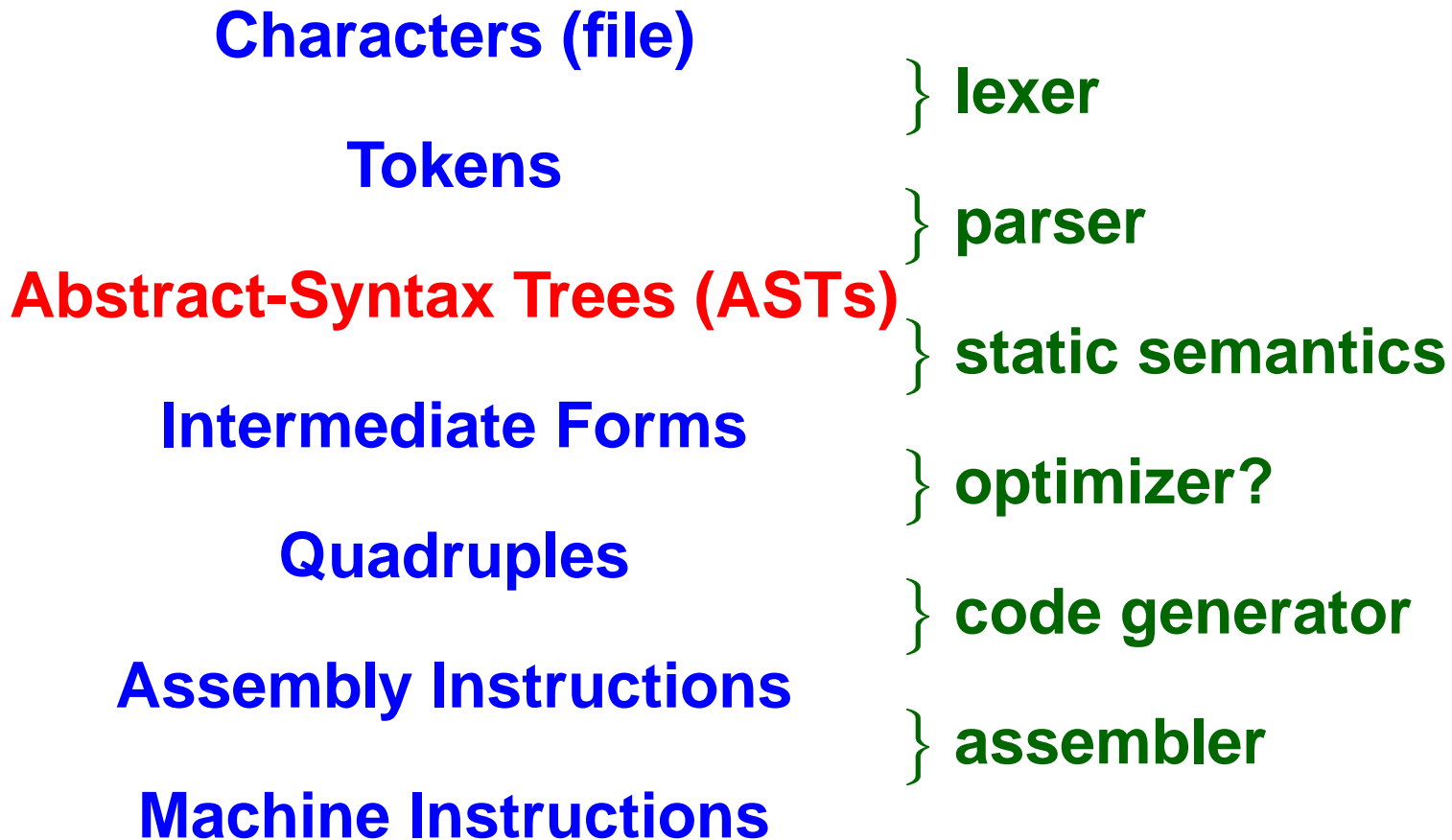
**normal form, reduction, eval order,
type systems, formal semantics**

More syllabus

<i>Unit/Language</i>	<i>Concepts</i>
CLU	abstract datatypes
Smalltalk	object-oriented programming
Standard ML	modules, exceptions
Prolog	logic programming, unification

Part I: Basics

Forms and translation of programs



Supporting cast

Compile time

- **Symbol table** (implements *environments*)

Run time

- **Instrumentation**
 - profiling, tracing, testing
- **Run-time system**
 - dynamic typing
 - **memory management**
 - exceptions
- **Debugging**

An Imperative Core

Models heart of most languages

Trivial syntax (**syntax** \in **CS 153**)

parenthesized prefix expressions (LISP-like)

Two kinds of inputs

- **function definitions**

```
(define mod (m n) (- m (* n (/ m n))))
```

like the C function

```
int mod (int m, int n) {  
    return m - n * (m / n);  
}
```

- **expressions**

An expression-oriented language

Expressions include control flow (no “statements”)

<code>(if e1 e2 e3)</code>	<code>if (e1) e2; else e3;</code>
<code>(while e1 e2)</code>	<code>while (e1) e2; 0</code>
<code>(set x e)</code>	<code>x = e</code>
<code>(begin e1 ... en)</code>	<code>(e1; ... ; en)</code>
<code>(f e1 ... en)</code>	<code>f(e1, ... , en)</code>

`f` may be primitive or defined with `(define f ...)`
primitives include:

`+ - * / = < > print`

More Impcore

Datatypes: **integer**

(we have functions, but they aren't values)

Scopes (name spaces, environments):

2 levels only: globals, formals

no local variables:

use excess formals (as in awk)

will fix for homework

functions live in their own name space

(not shared with variables)

Separate name spaces at work

```
-> (val f 33)
```

```
33
```

```
-> (define f (x) (+ x x))
```

```
f
```

```
-> (f f)
```

```
66
```

Impcore concrete syntax

toplevel ⇒ *expression* | *fundef* | *val-binding* | (use *filename*)

fundef ⇒ (define *function-name formals expression*)

formals ⇒ ({ *parameter-name* })

val-binding ⇒ (val *variable-name expression*)

expression ⇒ *literal-value*

| *variable-name*

| (if *expression expression expression*)

| (while *expression expression*)

| (set *variable-name expression*)

| (begin *expression* { *expression* })

| (*op* { *expression* })

op ⇒ *function-name* | + | - | * | / | = | < | > | print

More syntax

literal-value ⇒ *integer*

integer ⇒ digits, with optional - sign

**-name* ⇒ characters, but not () ; or blank

Abstract syntax

Input translated into efficient internal representation: **abstract-syntax tree** **exp**

LITERAL (integer)

VAR (name)

SET (name, exp)

IFX (exp, exp, exp)

WHILEX (exp, exp)

BEGIN (explist)

APPLY (name, explist)

**Both built-in and user-defined functions are
“application”**

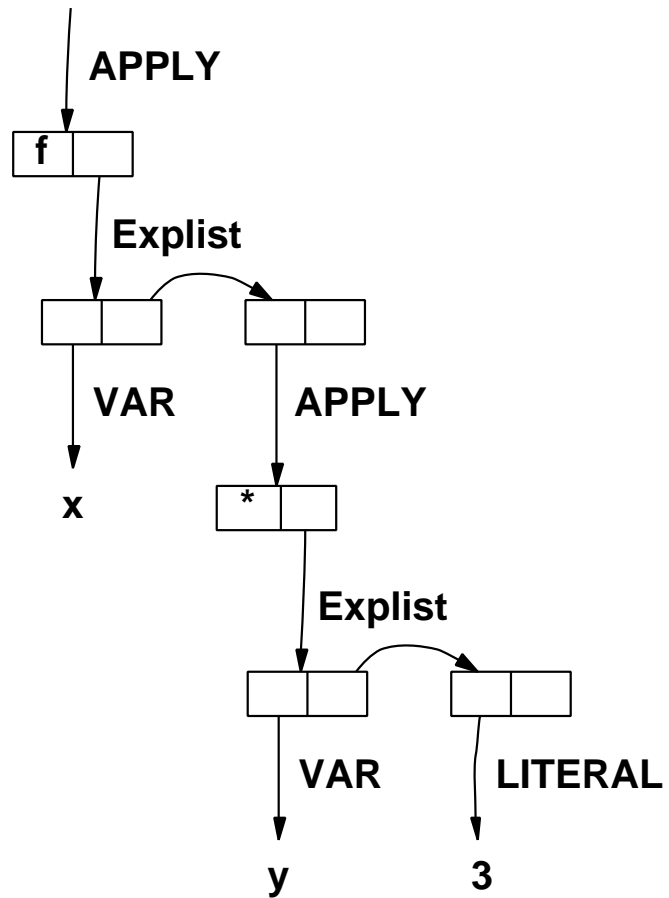
Representing abstract syntax

A recursive type: typedef struct Exp Exp;

```
struct Exp {
    enum
    { LITERAL, VAR, SET, IFX, WHILEX, BEGIN, APPLY } ty;
    union {
        Value literal;
        Name *var;
        struct { Name *name; Exp *exp; } set;
        struct { Exp *cond; Exp *true; Exp *false; } ifx;
        struct { Exp *cond; Exp *exp; } whilex;
        Explist *begin;
        struct { Name *name; Explist *actuals; } apply;
    } u;
};
```

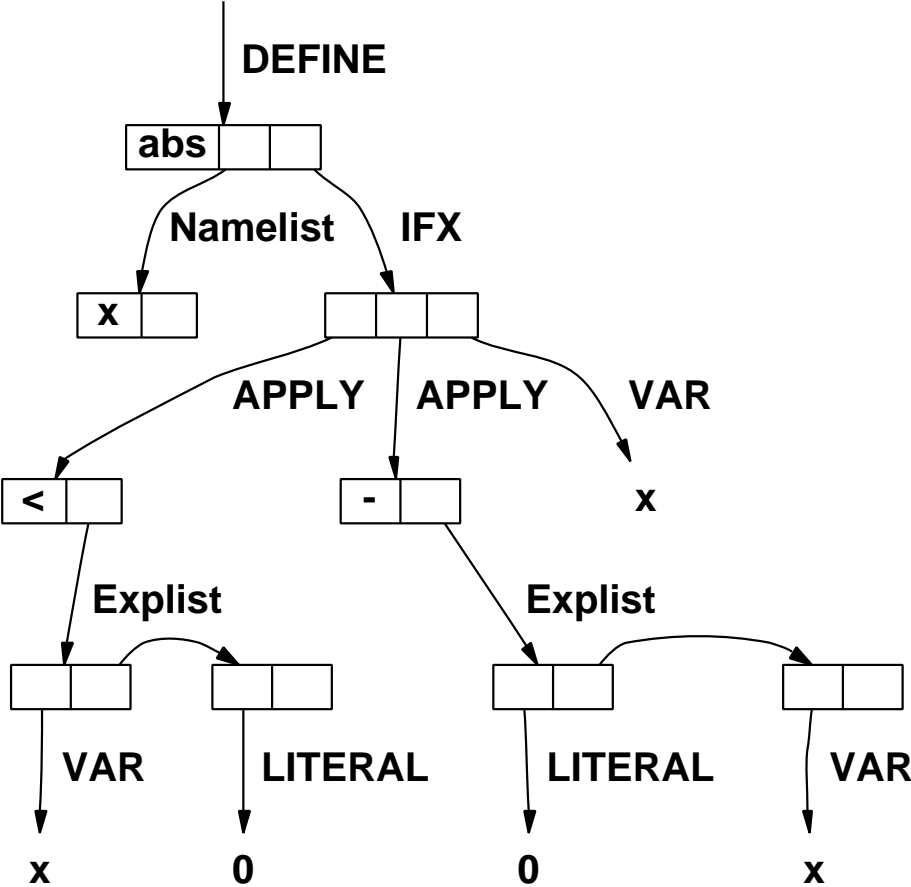
Example AST

Given input (f x (* y 3)) :



Another AST

```
(define abs (x) (if (< x 0) (- 0 x) x)):
```



Meanings, part I: names

Focus on *environments*:

associate values with variables

Environment ρ is mapping $\{x_1 \mapsto n_1, \dots, x_k \mapsto n_k\}$,
which associates variables x_i with values n_i .

$\rho(x)$ denotes value associated with variable x in
environment ρ

Environments as abstract type

Declaration:

```
typedef struct Valenv Valenv;
```

Constructor:

```
Valenv *mkValenv(Namelist *vars, Valuelist *vals);
```

Observers:

```
int isvalbound(Name *name, Valenv *env);
```

```
Value fetchval(Name *name, Valenv *env);
```

Mutator:

```
void bindval(Name *name, Value val, Valenv *env);
```

Implementing environments

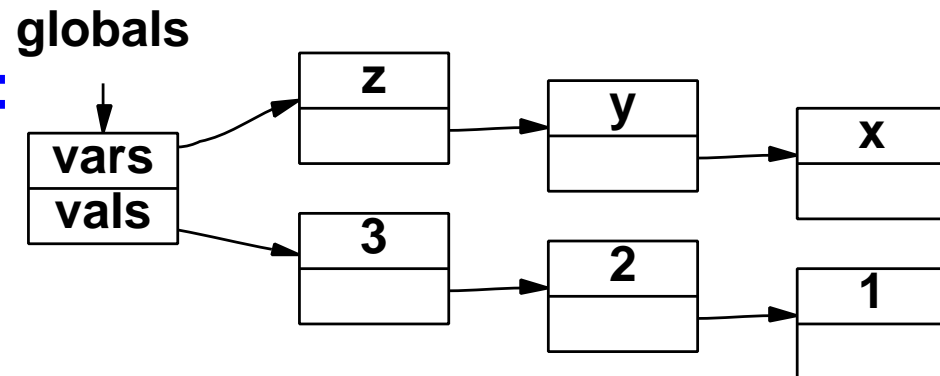
Use pair of lists, e.g., after

```
(set x 1)
```

```
(set y 2)
```

```
(set z 3)
```

global environment:



Desire for efficient representation, allocation, & deallocation of environments often drives language design (e.g., call stack—Exercise 13)

Meanings, part II: expressions

Expressions **evaluated** w.r.t. environment
(composition of formal, global, function environments)

Heart of the interpreter

- **structural recursion** on `ExpS`
- **environment provides meanings of names**

How do we explain evaluation?

Answer three questions:

1. What are the expressions?
2. What are the values?
3. What are the rules for turning expressions into values?

Combined: *operational semantics*

Operational semantics

Specify **executions** of programs on an **abstract machine**

Typical uses

- Very concise and precise language definition
- Direct guide to implementor
- Prove things like “**well-typed programs don’t go wrong**”

Operational Semantics

Loosely speaking, an interpreter

More precisely, formal rules for interpretation

- Set of **expressions**, also called **terms**
- Set of **values**
- **Full state of abstract machine**
(e.g., $\langle e, \xi, \phi, \rho \rangle$, \equiv **expression + 3 environments**)
- **Well specified initial state**
- **Transition rules for the abstract machine**
 - Good programs end in an **accepting state**
 - Bad programs **get stuck** (\equiv “go wrong”)

Operational semantics for Impcore

You've seen expressions: **ASTs**

All values are integers.

State $\langle e, \xi, \phi, \rho \rangle$ is

e Expression being evaluated

ξ Values of global variables

ϕ Definitions of functions

ρ Values of formal parameters

Rules form a **proof system** for judgment:

$$\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$$

Impcore semantics: Literals

$\overline{\langle \text{LITERAL}(\nu), \xi, \phi, \rho \rangle} \Downarrow \langle \nu, \xi, \phi, \rho \rangle$

(LITERAL)

Impcore semantics: Variables

Parameters hide global variables.

$$\frac{x \in \text{dom } \rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \quad (\text{FORMALVAR})$$

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle} \quad (\text{GLOBALVAR})$$

Impcore semantics: Assignment

$$\frac{x \in \text{dom } \rho \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle \nu, \xi', \phi, \rho' \rangle}{\langle \mathbf{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle \nu, \xi', \phi, \rho' \{x \mapsto \nu\} \rangle}$$

(FORMALASSIGN)

$$\frac{x \notin \text{dom } \rho \quad x \in \text{dom } \xi \quad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle \nu, \xi', \phi, \rho' \rangle}{\langle \mathbf{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle \nu, \xi' \{x \mapsto \nu\}, \phi, \rho \rangle}$$

(GLOBALASSIGN)

Evaluation code

```
Value eval(Exp *e,  $\xi$ ,  $\phi$ ,  $\rho$ ) {
    switch(e->ty) {
    case LITERAL: return e->u.literal;
    case VAR: ... /* look up in  $\rho$  and  $\xi$  */
    case SET: ... /* modify  $\rho$  or  $\xi$  */
    case IFX: ...
    case WHILEX: ...
    case BEGIN: ...
    case APPLY: f = fetchfun(e->u.apply.name,  $\phi$ );
                ... /* user fun or primitive */
    }
}
```

Evaluation cases

- VAR** find binding for variable and use value
- SET** rebind variable in `formals` or `globals`
- IFX** (recursively) evaluate condition, then `t` or `f`
- WHILEX** (recursively) evaluate condition, body
- BEGIN** (recursively) evaluate each `Exp` of body
- APPLY** look up function in `functions`
 - built-in PRIMITIVE** — do by cases
 - USERDEF function** —
 - use arg values to build `formals env`
 - recursive `eval` using fun body

Evaluation — Variables

To evaluate x , find binding $\rho(x)$, get value

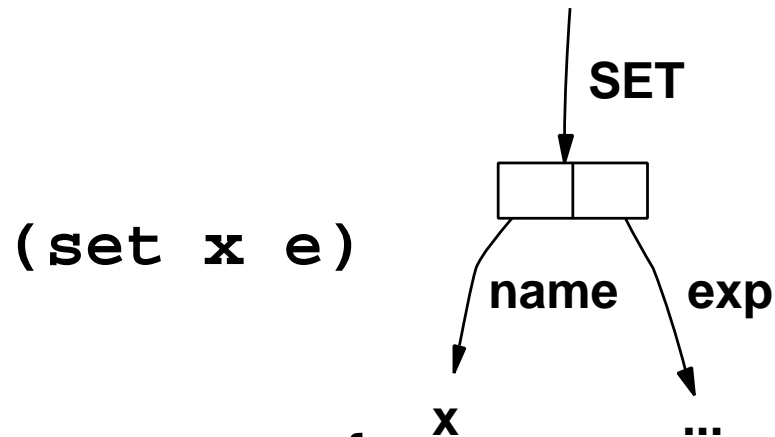
Conceptually, *one* environment, composed of
formals+globals

Composition implemented in `eval`, not in `Env` type:

case VAR:

```
if (isvalbound(e->u.var, formals))
    return fetchval(e->u.var, formals);
else if (isvalbound(e->u.var, globals))
    return fetchval(e->u.var, globals);
else
    error("unbound variable %n", e->u.var);
```

Assignment



Means change $\rho(x)$:
change parameter or
change global

```
case SET: {
```

```
    Value v = eval(e->u.set.exp, globals, functions, formals);
```

```
    if(isvalbound(e->u.set.name, formals))
```

```
        bindval(e->u.set.name, v, formals);
```

```
    else if(isvalbound(e->u.set.name, globals))
```

```
        bindval(e->u.set.name, v, globals);
```

```
    else
```

```
        error("set: unbound variable %n", e->u.set.name);
```

```
    return v; }
```

Evaluation — Application

1. Find function in old environment

```
f = fetchfun(e->u.apply.name, functions);
```

2. Evaluate actuals to get list of values (also in old ρ)

```
v1 = evallist(e->u.apply.actuals, globals, functions, formals);
```

N.B. actuals evaluated in the current environment

3. Make new env, binding formals to actuals

```
new_formals = mkValenv(f.u.userdef.formals, v1);
```

4. Evaluate body in new environment

```
return eval(f.u.userdef.body, globals, functions, new_formals);
```

Application — binding parameters

Actuals evaluated in the current environment

Result is `ValueList` — “half of an environment”
(reason why pair of lists, not list of pairs)

Formals are bound to actuals in a new environment
`mkValenv` builds an environment from two lists

Application semantics

$$\phi(f) = \mathbf{USER}(\langle x_1, \dots, x_n \rangle, e)$$

x_1, \dots, x_n all distinct

$$\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle$$

\vdots

$$\langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle$$

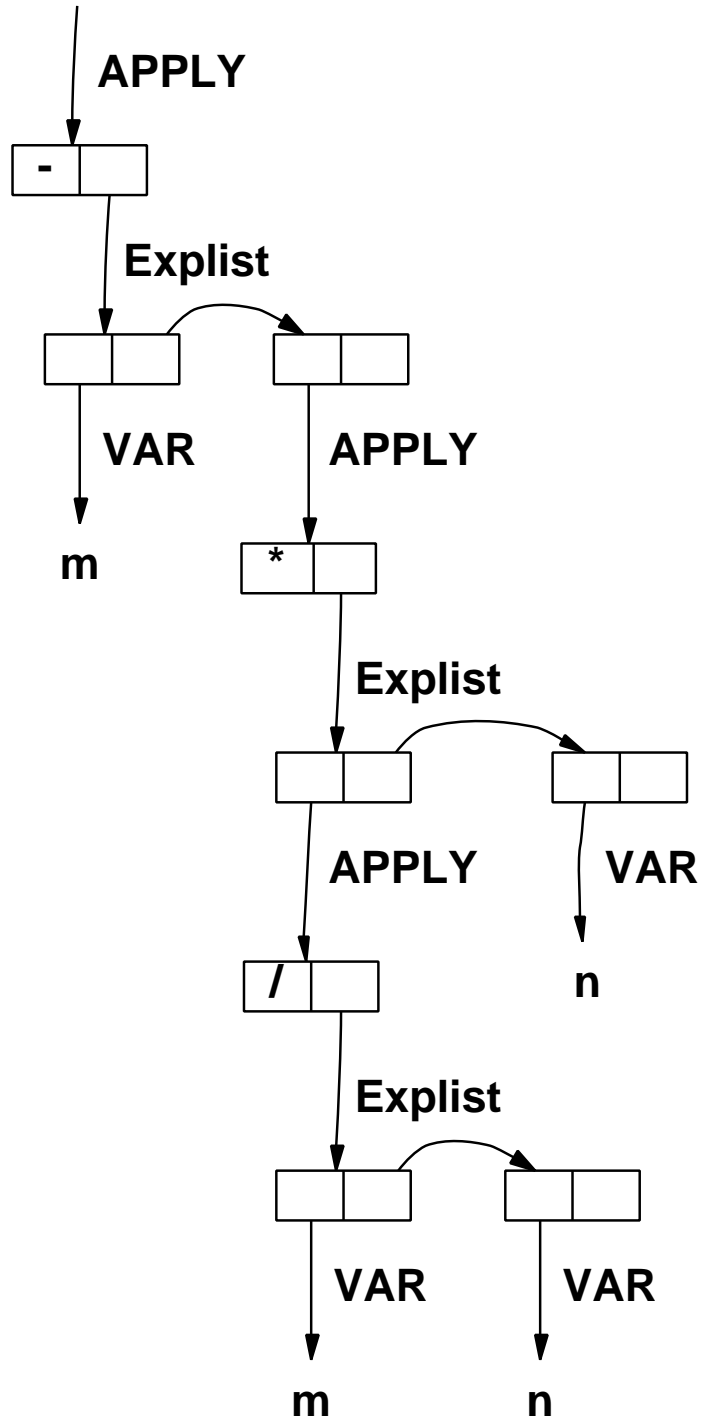
$$\langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$$

$$\frac{\langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \mathbf{APPLY}(f, e_1, \dots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle}$$

(APPLYUSER)

Application Example

```
(define mod (m n)
  (- m (* (/ m n) n)))
```



Things to notice about Impcore

Lots of environments:

global variables

functions

parameters

local variables?

More environments = more name spaces

⇒ more complexity

Typical of many programming languages.

Questions to remember

Abstract syntax: what are the terms?

Values: what do terms evaluate to?

Environments: what can names stand for?

Evaluation rules: how to evaluate terms?

Initial basis (primitives+): what's built in?